Iteration-Fusing Conjugate Gradient

Sicong Zhuang
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
sicong.zhuang@bsc.es

Marc Casas
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
marc.casas@bsc.es

ABSTRACT

This paper presents the Iteration-Fusing Conjugate Gradient (IFCG) approach which is an evolution of the Conjugate Gradient method that consists in i) letting computations from different iterations to overlap between them and ii) splitting linear algebra kernels into subkernels to increase concurrency and relax data-dependencies. The paper presents two ways of applying the IFCG approach: The IFCG1 algorithm, which aims at hiding the cost of parallel reductions, and the IFCG2 algorithm, which aims at reducing idle time by starting computations as soon as possible. Both IFCG1 and IFCG2 algorithms are two complementary approaches aiming at increasing parallel performance. Extensive numerical experiments are conducted to compare the IFCG1 and IFCG2 numerical stability and performance against four state-of-the-art techniques. By considering a set of representative input matrices, the paper demonstrates that IFCG1 and IFCG2 provide parallel performance improvements up to 42.9% and 41.5% respectively and average improvements of 11.8% and 7.1% with respect to the best state-of-the-art techniques while keeping similar numerical stability properties. Also, this paper provides an evaluation of the IFCG algorithms' sensitivity to system noise and it demonstrates that they run 18.0% faster on average than the best state-of-the-art technique under realistic degrees of system noise.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**;

KEYWORDS

Sparse Linear Algebra, Overlap Between Iterations, Mitigation of Synchronization Costs, Task Parallelism

1 INTRODUCTION

Many relevant High Performance Computing (HPC) applications have to deal with linear systems derived from using discretization schemes like the finite differences or finite elements methods to solve Partial Differential Equations (PDE). Typically, such discretization schemes produce large matrices with a significant degree of sparsity. Direct methods like the LU or the QR matrix factorizations are not applicable to such large matrices due to the significant number of steps they require to fully decompose them. Iterative methods are a much better option in terms of computational cost and, in particular, Krylov subspace methods are among the most successful ones. The basic idea behind Krylov methods when solving a linear system $Ax = b$ is to build a solution within the Krylov subspace composed of several powers of matrix A multiplied by vector b, that is, $\{b, Ab, A^2b, ..., A^mb\}$.

The fundamental linear operations involved in Krylov methods are the sparse matrix-vector (SpMV) product, the vector-vector addition and the dot-product. The performance of the sparse matrix-vector product is strongly impacted by irregular memory access patterns driven by the irregular positions of the sparse matrix's non-zero coefficients. As such, SpMV is typically an expensive memory-bound operation that benefits from large memory bandwidth capacity and also from high-speed interconnection networks. The vector-vector additions involved in Krylov methods typically have strided and regular memory access patterns and benefit a lot from resources like memory bandwidth and mechanisms like hardware pre-fetching to enhance their performance. Finally, the dot-product kernels involve expensive parallel operations like global communications and reductions that constitute an important performance bottleneck when running Krylov subspace methods [19].

Taking into account the performance aspects of the most fundamental linear algebra kernels of Krylov subspace methods, there are some natural improvements that are described in detail in the literature. For example, reducing the number of global reductions required by Krylov methods is a well-known option [3, 27]. Indeed, several variations of the Conjugate Gradient (CG) algorithm have been suggested to reduce the number of global dot-products to just one [10, 13, 23, 28]. There is also work focused on reducing the number of global synchronizations targeting other subspace Krylov methods (e. g. BiCG and BiCGStab) [9, 17, 30, 31]. S-step Krylov methods also aim at reducing the number of global reductions [2, 14, 21, 22]. Besides reducing the number of global synchronization points, another alternative to boost Krylov subspace methods performance is to overlap the two most expensive kernels (SpMV and dot-product) either with other computations or between them. Indeed, overlapping the two dot-products of the CG algorithm with the residual update has already been proposed [15], as well as an asynchronous version of the CG algorithm to overlap one of the global reductions with the SpMV and the other with the preconditioner [18]. A variant of the CG algorithm that performs the two global reductions per iteration at once and also hides its latency by overlapping them with the SpMV kernel has also been proposed [26]. Despite this extensive body of work devoted on improving the CG algorithm, performance enhancements brought by state-of-the-art approaches are still far from providing good scalability results [26].

In this work we propose the *Iteration-Fusing Conjugate Gradient (IFCG)* method, a new formulation of the CG algorithm that outperforms the existing proposals by applying a scheme that aggressively overlaps iterations, which is something not considered by previous work. Our approach does not update the residual at the end of each iteration and splits numerical kernels into subkernels to relax data-dependencies. By carrying out these two optimizations, our approach allows computations belonging to different iterations to overlap if there are no specific data or control dependencies between them. This paper provides two algorithms that implement

the IFCG concept: IFCG1, which aims at hiding the costs of global synchronizations and IFCG2, which starts computations as soon as possible to avoid idle time.

From the programming perspective, there are several ways to enable the overlap of different pieces of computation during a parallel run. For example, such overlaps can be expressed at the parallel application source code level by using sophisticated programming techniques like pools of threads or asynchronous calls [4]. Other alternatives conceive the parallel execution as a directed graph where nodes represent pieces of code, which are named tasks, and edges represent control or data dependencies between them. Such approaches require the programmer to annotate the source code in order to express such dependencies and let a runtime system orchestrate the parallel execution. In this way, the maximum available parallelism is dynamically extracted without the need for specifying suboptimal overlaps at the source code level. Approaches based on tasks are becoming important in the parallel programming area. Indeed, commonly used shared memory programming models like OpenMP include advanced tasking constructs [25] and it is also possible to run task-based workloads on distributed memory environments [6].

This paper adopts this task-based paradigm and applies it to the IFCG1 and IFCG2 parallel algorithms. Specifically, this paper improves the state-of-the-art by doing the following contributions:

- The Iteration-Fusing Conjugate Gradient (IFCG) approach, which aims at aggressively overlapping different iterations. IFCG is implemented by means of two algorithms: IFCG1 and IFCG2.
- A task-based implementation of the IFCG1 and IFCG2 algorithms that automatically overlaps computations from different iterations without the need for explicit programmer specification on what computations should be overlapped.
- A comprehensive evaluation comparing IFCG1 and IFCG2 with the most relevant state-of-the-art formulations of the CG algorithm: Chronopoulos' CG [10], Gropp's CG [18], Pipelined CG [26] and a basic Preconditioned CG method. All 6 CG variants are implemented via a task-based programming model to provide a fair evaluation. IFCG1 and IFCG2 provide parallel performance improvements up to 42.9% and 41.5% respectively and average improvements of 11.8% and 7.1% with respect to the state-of-the-art techniques and show similar numerical stability.
- A demonstration that under realistic system noise regimes IFCG algorithms behave much better than previous approaches. IFCG algorithms achieve an average 18.0% improvement over the best state-of-the-art techniques under realistic system noise regimes.

This paper is structured as follows: In section 2 we describe in detail some state-of-the-art approaches that motivate the IFCG algorithms. Section 3 contains a detailed description of the IFCG1 and IFCG2 algorithms. Section 4 compares the numerical stability of IFCG1 and IFCG2 with other relevant state-of-the-art techniques. Section 5 explains how task-based parallelism is applied to IFCG1 and IFCG2 and how they are executed in parallel. Section 6 describes in detail the experimental setup of this paper. Section 7 shows a comparison of IFCG1 and IFCG2 against other state-of-the-art techniques when run on a 16-core node composed of two 8-core sockets. It also discusses other important aspects like the inter-iteration overlap achieved by IFCG1 and IFCG2 and a comparison of the system jitter tolerance of these algorithms against state-of-the-art approaches. Finally, section 8 contains several conclusions on this work and describes future directions.

2 THE PRECONDITIONED AND THE PIPELINED CG ALGORITHMS

We describe the basic Preconditioned Conjugate Gradient Algorithm and one of its most important evolutions, the Pipelined Conjugate Gradient [26], which aims at improve CG's performance by reducing the cost of its global reductions.

2.1 Preconditioned Conjugate Gradient

The fundamental Preconditioned Conjugate Gradient (PCG) algorithm is a Krylov subspace method that iteratively builds a solution in terms of a basis of conjugate vectors built by projecting the maximum descent direction, i.e. the gradient, into the closest conjugate direction. PCG is shown in **Algorithm 1**. Performance-wise, steps 4 and 8 are important bottlenecks since they involve a global reduction. Pre-conditioning the vector r_{i+1} (carried out by step 7) is also typically expensive.

Algorithm 1 PCG

1: $r_0 := b - Ax_0; u_0 := M^{-1}r_0; p_0 := u_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: $\quad s := Ap_i$
4: $\quad \alpha := (r_i, u_i)/(s, p_i)$
5: $\quad x_{i+1} := x_i + \alpha p_i$
6: $\quad r_{i+1} := r_i - \alpha s$
7: $\quad u_{i+1} := M^{-1}r_{i+1}$
8: $\quad \beta := (r_{i+1}, u_{i+1})/(r_i, u_i)$
9: $\quad p_{i+1} := u_{i+1} + \beta p_i$
10: **end for**
11: Inter-iteration synchronization

2.2 Pipelined Conjugate Gradient

The Pipelined Conjugate Gradient (Pipelined CG) [26] is an alternative formulation of the PCG algorithm aiming at i) reducing the cost of the two PCG's reduction operations per iteration by concentrating them into a single double reduction point and ii) hiding the cost of this double reduction by overlapping it with other PCG kernels: SpMV and the preconditioner. The Pipelined CG formulation is mathematically equivalent to PCG and, indeed, both techniques would give the exact same solution if they operated with infinite precision. However, when operating under realistic scenarios, i.e. 32- or 64-bits floating point representations, Pipelined CG exhibits worse numerical accuracy than PCG since the way Pipelined CG builds the basis of conjugate vectors is more sensitive to round-off errors, which ends up having an impact on the basis' orthogonality.

The Pipelined CG technique is detailedly shown in **Algorithm 2**. The two reductions are computed at the beginning of each iteration (lines 3-4), which makes it possible to combine them. Additionally, this double reduction operation is overlapped with two costly computations: the application of the preconditioner to vector w_i (line 5) and a sparse matrix-vector product (line 6). It is important to state that, although the potential of Pipelined CG in terms of

Algorithm 2 Pipelined CG

1: $r_0 := b - Ax_0; u_0 := M^{-1}r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: $\gamma_i := (r_i, u_i)$
4: $\delta_i := (w_i, u_i)$
5: $m_i := M^{-1}w_i$
6: $n_i := Am_i$
7: **if** $i > 0$ **then**
8: $\beta_i := \gamma_i/\gamma_{i-1}; \alpha_i := \gamma_i/(\delta_i - \beta_i\gamma_i/\alpha_{i-1})$
9: **else**
10: $\beta_i := 0; \alpha_i := \gamma_i/\delta_i$
11: **end if**
12: $z_i := n_i + \beta_i z_{i-1}$
13: $q_i := m_i + \beta_i q_{i-1}$
14: $s_i := w_i + \beta_i s_{i-1}$
15: $p_i := u_i + \beta_i p_{i-1}$
16: $x_{i+1} := x_i + \alpha_i p_i$
17: $r_{i+1} := r_i - \alpha_i s_i$
18: $u_{i+1} := u_i - \alpha_i q_i$
19: $w_{i+1} := w_i - \alpha_i z_i$
20: **end for**
21: Inter-iteration synchronization

overlapping computations and hiding reduction costs is remarkable, the algorithm still has limitations. For example, the update of the z vector in line 12 needs the whole n_i vector and the β_i scalar to be carried out. However, such restriction can be relaxed by breaking down the z update into several pieces that only have to wait for their n counterpart and the β_i scalar to be carried out. In this way, some pieces of the z vector can be updated without the need for waiting until the whole n_i vector is produced.

3 ITERATION-FUSING CONJUGATE GRADIENT

In this section we present the Iteration-Fusing Conjugate Gradient (IFCG) approach, which breaks down some of the Pipelined CG computations into smaller pieces to relax data dependencies and reduce idle time. Also, IFCG enables the overlap of different iterations by removing inter-iteration barrier points. We present two algorithms that implement the IFCG approach: The first one (IFCG1) improves the Pipelined CG formulation by letting different iterations to overlap as much as possible while the second one (IFCG2) aims at increasing performance even more by splitting Pipelined CG's single synchronization point into two and exploiting additional opportunities to reduce idle time. While both IFCG1 and IFCG2 algorithms apply the IFCG approach, they aim at increasing performance by targeting different goals.

3.1 IFCG1 Algorithm

IFCG1 is an evolution of the Pipelined CG algorithm described in section 2.2. IFCG1 aims at increasing the potential for overlapping different pieces of computation by breaking down the Pipelined CG kernels into smaller pieces or subkernels. Each subkernel just needs a subset of the data required by the whole kernel. For example, as mentioned a few paragraphs above, the update of the z vector in line 12 of **Algorithm 2** requires the whole n_i vector and the β_i scalar. Instead of considering the update of z as a single operation, IFCG1 breaks it down into N pieces in a way that instead of computing the whole $z_i := n_i + \beta_i z_{i-1}$ it computes N updates of the form $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$ where $z_i = \{z_{i1}, z_{i2}, ..., z_{iN}\}$ and i refers to the ith iteration. In this way, the computation of z_{ij} only depends on a subset n_{ij} of the n_i vector and the scalar

Algorithm 3 IFCG1

1: $r_0 := b - Ax_0; u_0 := M^{-1}r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: **for** $j = 1 \ldots N$ **do** ▷ The computation is split in N blocks
4: $\gamma_{ij} := (r_{ij}, u_{ij})$
5: $\delta_{ij} := (w_{ij}, u_{ij})$
6: $m_{ij} := M^{-1}w_{ij}$
7: $n_{ij} := A_j m_i$
8: **end for**
9: $\gamma_i := \sum_{j=1}^{N} \gamma_{ij}; \delta_i := \sum_{j=1}^{N} \delta_{ij}$ ▷ Global reduction
10: **if** $i > 0$ **then**
11: $\beta_i := \gamma_i/\gamma_{i-1}; \alpha_i := \gamma_i/(\delta - \beta_i\gamma_i/\alpha_{i-1})$
12: **else**
13: $\beta_i := 0; \alpha_i := \gamma_i/\delta_i$
14: **end if**
15: **for** $j = 1 \ldots N$ **do**
16: $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$
17: $q_{ij} := m_{ij} + \beta_i q_{(i-1)j}$
18: $s_{ij} := w_{ij} + \beta_i s_{(i-1)j}$
19: $p_{ij} := u_{ij} + \beta_i p_{(i-1)j}$
20: $x_{(i+1)j} := x_{ij} + \alpha_i p_{ij}$
21: $r_{(i+1)j} := r_{ij} - \alpha_i s_{ij}$
22: $u_{(i+1)j} := u_{ij} - \alpha_i q_{ij}$
23: $w_{(i+1)j} := w_{ij} - \alpha_i z_{ij}$
24: **end for**
25: **end for**

β_i. The only operation that can not always be broken down into pieces is the computation of the preconditioning vector w_i (step 5 of **Algorithm 2**). While some preconditioning schemes can be decomposed into pieces (e. g. Block-Jacobi preconditioning with incomplete Cholesky factorization within the blocks) some others do not admit a straightforward decomposition (e. g. multi-grid preconditioning), although preconditioners that can be decomposed are often applied [26].

Besides breaking down linear algebra kernels into pieces, the second innovative aspect of the IFCG approach is the elimination of inter-iteration synchronizations to check algorithm's convergence. Instead of checking for convergence at the end of each iteration, IFCG only checks it once every n iterations. The number of iterations between two checks is called the *FUSE* parameter.

We apply these two approaches (decomposition of linear kernels and elimination of inter-iterations checks) across the whole Pipelined CG algorithm, which ends up producing the IFCG1 algorithm (**Algorithm 3**). IFCG1 can potentially overlap steps 4-7 of iteration i with steps 16-23 of iteration $i-1$. Also, each repetition of steps 16-23 depends on just one of the N repetitions of steps 6-7, significantly relaxing data-dependencies between the algorithm's main kernels.

3.2 IFCG2 Algorithm

The IFCG2 algorithm splits Pipelined CG and IFCG1's single synchronization point, which is composed of two reductions, into two synchronization points composed of a single reduction operation each. IFCG2 aims at updating the s_i and p_i vectors, which only depend on one of the two reductions and on some data generated by iteration $i-1$, as soon as possible. The IFCG2 algorithm is detailedly shown in **Algorithm 4**. The global reductions producing δ_i and γ_i are run in separate steps. Also, the updates on vectors s_i and p_i do not need to wait for the reduction producing δ_i to finish as they can be overlapped with it. Computing q_i and n_i is left after the second reduction since these computations require m_i and we want

Algorithm 4 IFCG2

1: $r_0 := b - Ax_0; u_0 := M^{-1}r_0; w_0 := Au_0$
2: **for** $i = 0 \ldots i_{max}$ **do**
3: **for** $j = 1 \ldots N$ **do**
4: $\gamma_{ij} := (r_{ij}, u_{ij})$
5: $\delta_{ij} := (w_{ij}, u_{ij})$
6: $m_{ij} := M^{-1}w_{ij}$ ▷ The most expensive step
7: **end for**
8: $\gamma_i := \sum_{j=1}^{N} \gamma_{ij}$ ▷ Global reduction on γ_i
9: **if** $i > 0$ **then**
10: $\beta_i := \gamma_i / \gamma_{i-1}$
11: **else**
12: $\beta_i := 0$
13: **end if**
14: **for** $j = 1 \ldots N$ **do** ▷ AXPYs that only depend on β_i
15: $s_{ij} := w_{ij} + \beta_i s_{(i-1)j}$
16: $p_{ij} := u_{ij} + \beta_i p_{(i-1)j}$
17: **end for**
18: $\delta_i := \sum_{j=1}^{N} \delta_{ij}$ ▷ Global reduction on δ
19: **if** $i > 0$ **then**
20: $\alpha_i := \gamma_i / (\delta_i - \beta_i \gamma_i / \alpha_{i-1})$
21: **else**
22: $\alpha_i := \gamma_i / \delta_i$
23: **end if**
24: **for** $j = 1 \ldots N$ **do**
25: $q_{ij} := m_{ij} + \beta_i q_{(i-1)j}$
26: $n_{ij} := A_j m_i$
27: $z_{ij} := n_{ij} + \beta_i z_{(i-1)j}$
28: $x_{(i+1)j} := x_{ij} + \alpha_i p_{ij}$
29: $r_{(i+1)j} := r_{ij} - \alpha_i s_{ij}$
30: $u_{(i+1)j} := u_{ij} - \alpha_i q_{ij}$
31: $w_{(i+1)j} := w_{ij} - \alpha_i z_{ij}$
32: **end for**
33: **end for**

the reductions to be overlapped as much as possible with the most expensive computational kernel, the preconditioning of vector ω_i (step 6 of **Algorithm 4**).

There is an interesting trade-off between the IFCG1 and IFCG2 algorithms: While the first one is focused on reducing the cost of the two global reductions by overlapping them with computations, which implies delaying the update of the s_i and p_i vectors, the second tries to run these updates as soon as possible, which requires splitting the single synchronization point composed of two reductions into two parallel dot-products. As such, the IFCG1 formulation aims at reducing the cost of reduction operations while the IFCG2 aims at starting the computations as soon as possible to avoid idle time. IFCG1 and IFCG2 algorithms are thus two complementary approaches that constitute an evolution of the Pipelined CG algorithm aiming at increasing performance. Besides the parallel programming and performance aspects, which are detailedly discussed in sections 5 and 7, it is also important to verify that both IFCG algorithms have similar numerical stability properties as state-of-the-art approaches like Pipelined CG.

4 NUMERICAL STABILITY OF THE IFCG ALGORITHMS

The main issue with the numerical stability of IFCG algorithms is the same as the one displayed by many other Krylov-based methods: The way the residual vector r is computed. It is usually done by just updating the residual of iteration i from the one in iteration $i-1$ via expressions like $r_i = r_{i-1} - \alpha\, Ap_{i-1}$. However, by doing so,

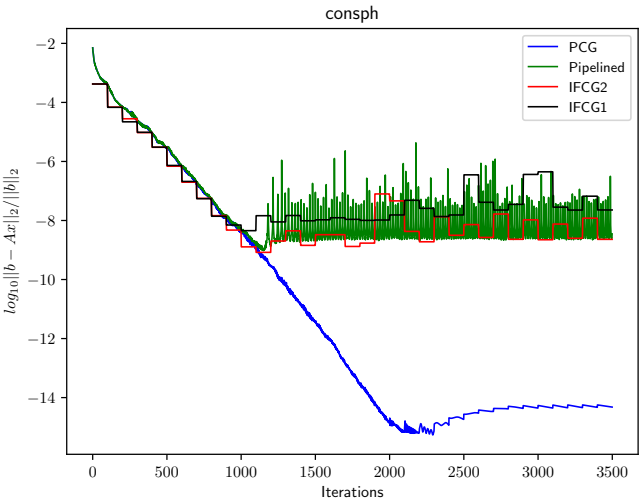

Figure 1: Convergence of the Preconditioned CG, Pipelined CG, IFCG1 and IFCG2 algorithms. Data regarding IFCG1 and IFCG2 is reported every 100 iterations since *FUSE* = 100.

residual r_i may deviate from the true residual $b - Ax_i$. State-of-the-art approaches use a residual replacement strategy to prevent the updated residual r_i to deviate from the true residual. The remedy is to periodically replace the updated r_i by $b - Ax_i$ [7, 26, 29]. The frequency of such replacement is a trade-off between convergence speed and accuracy and some sophisticated strategies exist [8, 29] to deal with it. In the case of IFCG and IFCG2 we do the $r_i = b - Ax_i$ replacement every *FUSE* iterations to avoid hurting the overlap between different iterations.

We run some experiments considering several sparse matrices obtained from the Florida Sparse Matrix Collection [12]. These experiments involve parallel executions of the IFCG1, IFCG2, Pipelined CG and Preconditioned CG algorithms on a 16 cores NUMA node composed of two 8-core sockets. More specific details on the parallel implementations and the precise experimental setup can be found in sections 5 and 6, respectively. Also, Table 1 contains a description of the matrices considered in the experiments.

Figure 1 displays the evolution of the relative residual $||b - Ax_i||_2/||b||_2$ on matrix *consph* considering the IFCG1, IFCG2, Pipelined CG and Preconditioned CG methods. Data concerning IFCG1 and IFCG2 are expressed in a coarser-grain pattern than Pipelined and Preconditioned CG's data points since we calculate the relative residual every 100 iterations (i. e. *FUSE*=100). We can see that the convergence of the IFCG1 and IFCG2 algorithms is the same as Pipelined CG. Interestingly, Figure 1 also displays how the basic Preconditioned CG algorithm has better convergence properties than Pipelined CG, which is consistent with previously reported numerical results [11, 26]. We omit results concerning the rest of the matrices in Table 1 since they exhibit the exact same behavior as the one observed with *consph*. Our experiments show how the numerical behavior in terms of the relative residual $||b-Ax_i||_2/||b||_2$ achieved by IFCG1 and IFCG2 matches the state-of-the-art.

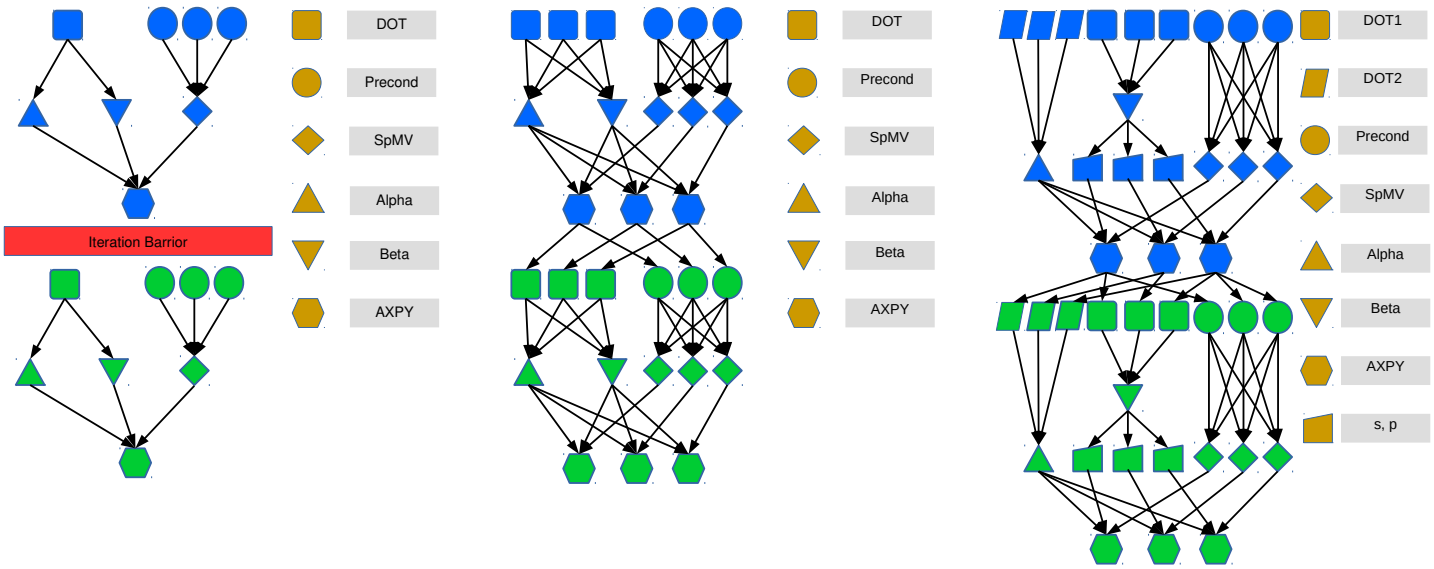

Figure 2: Graphs of tasks representing two Iterations of Pipelined CG (left), IFCG1 (center) and IFCG2 (right), $N = 3$.

5 PARALLEL EXECUTION OF THE IFCG ALGORITHMS

The IFCG algorithms have been carefully designed to hide the impact of their global synchronization points by overlapping them with other numerical kernels. Also, IFCG algorithms aim at relaxing data-dependencies between these different kernels by breaking them down into several subkernels that just require a reduced input data set to carry on. These features can significantly improve performance but, in order to exploit them, the IFCG algorithms must run in parallel and enforce the overlap of the different computational kernels as much as possible. Therefore, we need to specify at the source code level a parallel scheme that meets these requirements and there are several ways to do so.

One option is to statically specify at the application source code level the way different kernels overlap with each other, which should be done by means of sophisticated parallel programming techniques like pools of threads or active waiting loops that trigger work once its input data is ready. However, the optimality of these techniques depends a lot on the parallel hardware where the parallel execution takes place. Therefore, a static approach is not practical since it needs to be adapted to each parallel execution scenario. In this paper we follow a dynamic approach that conceives the parallel execution as a directed acyclic graph where the nodes represent pieces of code (also known as tasks) and the edges are control or data dependencies between them. This approach requires from the programmer to specify the pieces of code or tasks that run in parallel by means of annotations that contain their input or output dependencies. A runtime system orchestrates the parallel run by considering tasks' input or control dependencies and scheduling them into the available parallel hardware once all dependencies are satisfied. The most important shared-memory programming models, like OpenMP, have support for this kind of task-based parallelism and there is also support for running task-based workloads on distributed memory environments [6].

5.1 Task-based Formulations of the Pipelined CG and IFCG algorithms

The Pipelined CG, IFCG1 and IFCG2 algorithms can be easily formulated in terms of tasks by just looking at each one of the steps in algorithms 2, 3 and 4 and considering them tasks. Indeed, by means of the #pragma annotations provided by OpenMP it is possible to specify that each one of these steps is a task as well as which are its data dependencies. Control dependencies are typically expressed in terms of sentinels. Importantly, IFCG1 and IFCG2 have many more tasks per iteration than Pipelined CG. Indeed, steps 3-6 and 12-19 of Pipeline CG (**Algorithm 2**) are split into N substeps in **Algorithms 3 and 4**, which implies that we have N tasks in IFCG1 and IFCG2 per each Pipeline CG task. The only exception to this rule is the preconditioning step which is typically split depending on whether or not the chosen preconditioner allows a decomposition in terms of tasks.

Figure 2 shows two iterations of the Pipelined CG, IFCG1 and IFCG2 algorithms represented in terms of task graphs. Parameter N is equal to 3, which means that many of the Pipelined CG tasks appearing in the task graph are broken down into 3 tasks by the IFCG1 and IFCG2 methods. In the case of the Pipelined CG algorithm the task named *DOT* represents steps 3 and 4 of **Algorithm 2** while tasks named *Precond* represents step 5, which in this particular case is divided into several tasks. Tasks α and β represent the computations done within step 8. Finally, the task designated as *AXPY* represents steps 12-19. Similarly, the task graph representations of the IFCG1 and IFCG2 methods represent all the steps displayed by algorithms 3-4.

By comparing the center and the left hand side task graphs in Figure 2 we can observe how by removing the iteration barrier and breaking the computation routines into blocks we expose much more parallelism to the hardware. Indeed, Pipelined CG has a limited potential for overlapping tasks belonging to the same iteration and cannot overlap tasks from different iterations at all since its inter-iteration barrier prevents it from doing so. In contrast, IFCG1 displays a much more flexible parallel pattern that can easily overlap tasks belonging to different iterations. IFCG2, by further extracting two AXPYs operations (s, p) that only depend on β, is able to create even more concurrency. The implications and analysis of these varying level of parallelism shown by the different algorithms are explained in Section 7.

Name	Dimension	Nonzeros	Nonzeros%
G3_circuit	1585478	7660826	0.0003%
thermal2	1228045	8580313	0.0006%
ecology2	999999	4995991	0.0005%
af_shell8	504855	17579155	0.0068%
G2_circuit	150102	726674	0.003%
cfd2	123440	3085406	0.02%
consph	83334	6010480	0.087%

Table 1: Matrices used for experiments

6 EXPERIMENTAL SETUP

We conduct our parallel experiments on a 16-cores node composed of two 8-core Intel Xeon® processors E5-2670 at 2.6 GHz and a 20 MB L3 shared cache memory with SuSe Linux OS. All the algorithms we consider in the evaluation (IFCG1, IFCG2, Preconditioned CG, Pipelined CG [26], Chronopoulos CG [10] and Gropp CG [18]) are implemented using the OpenMP4.0 programming model running on top of the Nanos++ (v0.7a) parallel runtime system [5]. We use the Intel's MKL [1] library to compute the fundamental linear algebra kernels involved in our experiments. All the aforementioned algorithms are implemented with the Block-Jacobi preconditioner with incomplete Cholesky factorization within the blocks. The block size N is set to 64 throughout the experiments and the convergence threshold is $||b - Ax_i||_2/||b||_2 < 10^{-7}$

We consider 7 Symmetric and Positive Definite (SPD) matrices from The University of Florida Sparse Matrix Collection [12]. We use two matrices from circuit simulation problems (*G3_circuit* and *G2_circuit*), two matrices from unstructured Finite Element Method schemes (*thermal2*, *consph*), one matrix from material engineering problems (*af_shell8*) and one matrix from a computational fluid dynamics problem (*cfd2*). In Table 1 we show a more precise description of all considered matrices in terms of their dimensions and sparsity. The considered matrices cover a wide range of dimensions (from 72,000 up to 1,585,478 rows and columns) and sparsity degrees, which makes them representative of the typical problems faced by the CG method and its variants.

7 EVALUATION

In this section we provide a comprehensive evaluation of the IFCG1 and the IFCG2 algorithms and we compare them in terms of performance with the 4 state-of-the-art techniques mentioned in Section 6. We first carry out a sensitivity study of the *FUSE* parameter to determine its optimal value. We then compare the performance of IFCG1 and IFCG2 running with this optimal *FUSE* value against the 4 state-of-the-art methods mentioned above. We demonstrate that IFCG1 and IFCG2 achieve a significant degree of overlap between iterations, which provides them with much better performance results than their competitors. Finally, we compare the noise tolerance of IFCG1 and IFCG2 against other CG variants. We consider two different noise regimes, both of them close to realistic noise scenarios, and we demonstrate that the IFCG algorithms are much more tolerant to system noise than state-of-the-art approaches.

7.1 Optimizing the *FUSE* Parameter.

As explained in previous sections, by removing the convergence check at the end of each iteration and just checking for convergence once every *FUSE* algorithmic steps, we let computations to overlap across different iterations. However, the algorithm may keep running once the threshold is met since convergence is only checked once every several iterations, which has an impact over the total execution time. If this extra time is larger than the benefits obtained from increasing the overlap across iterations, IFCG1 and IFCG2 will perform poorly. On the contrary, if we restrict the *FUSE* Parameter too much, that is, if we check for convergence too often, the potential for overlap will be undermined.

In Figure 3 we show the impact of the *FUSE* parameter on the scalability of the IFCG1 algorithm when applied to the 8 matrices described in section 6. We consider the *FUSE* parameter to be 1, 5, 20, 50, 80, 100 and 200. For each matrix we show the speedup achieved by varying the *FUSE* value and running IFCG1 on 1, 2, 4, 8 and 16 cores over the execution with *FUSE* = 1 on 1 core. In the x-axis we represent the total number of cores involved in the parallel execution while in the y-axis we show the speedup achieved by each technique. The input matrices and the experimental setup are described in Section 6.

When running on a single core we achieve speedups of 1.12x, 1.12x and 1.09x over the *FUSE* = 1 configuration when *FUSE* is set to 5, 20 and 50, respectively. This modest speedups are due to the reduction of overheads brought by checks for convergence, i. e. computing $Ax_i - b$, which is done once every *FUSE* iterations. These small benefits decrease for large *FUSE* values due to the extra iterations the algorithm carries out. When the experiments are run on larger core counts the benefits of increasing the *FUSE* value are very significant. Indeed, we achieve average speedups of 10.44x, 11.15x and 10.99x when *FUSE* is set to 5, 20 and 50 and IFCG1 runs on 16 cores with respect to the sequential run with *FUSE* = 1. In general, the benefits of increasing *FUSE* stall at 20 and start to decline when *FUSE* reaches the 200 value. Matrix-wise, results are very consistent since IFCG1 reaches optimal or very close to optimal performance when *FUSE* = 20 for 5 matrices: *cfd2*, *ecology2*, *consph*, *G2_circuit* and *thermal2*. Just for the *af_shell8* and *G3_circuit* matrices the *FUSE* optimal value is different from 20 (80 in the first case and 5 in the second) although the speedups in these optimal points (12.5x and 11.33x respectively), are very close to the ones achieved by the *FUSE* = 20 configuration (11.62x and 10.38x). In general, a *FUSE* value of 20 is the best one for the IFCG1

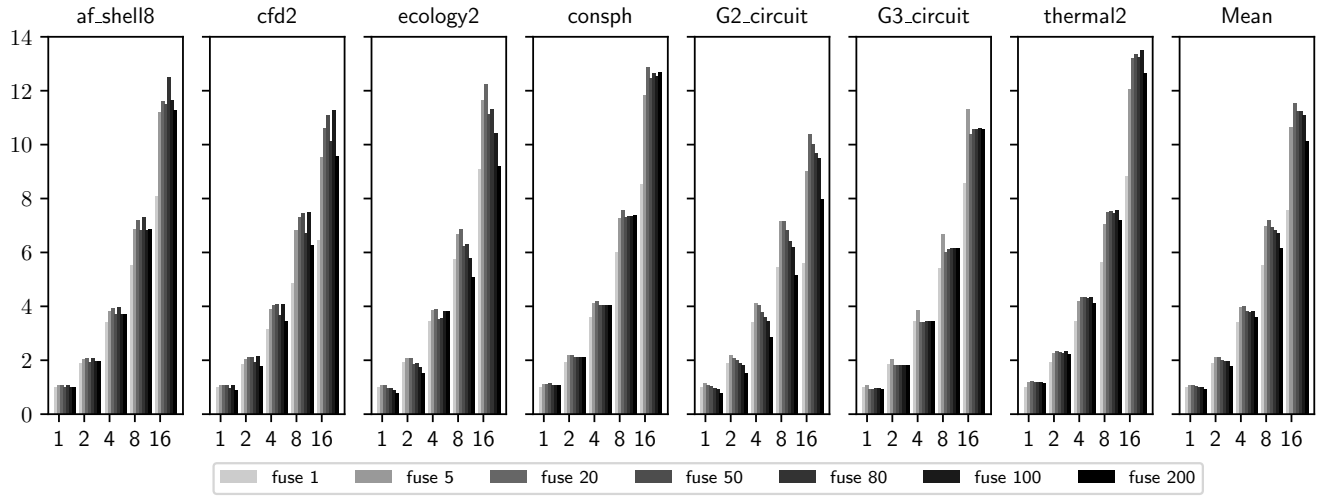

Figure 3: Impact of the *FUSE* parameter on IFCG1. The y-axis represents the achieved speedups with respect to the *FUSE*=1 configuration running on 1 core while x-axis represents core counts.

algorithm. By conducting the same analysis for IFCG2 we find its optimal *FUSE* parameter to be 20 as well.

7.2 Evaluation of the IFCG1 and IFCG2 algorithms against state-of-the-art techniques

This section provides an evaluation of the parallel speedups achieved by the IFCG1 and the IFCG2 algorithms and compares them with 4 state-of-the-art techniques: Preconditioned CG (PCG), Pipelined CG [26], Chronopoulos CG [10] and Gropp CG [18]. Both IFCG1 and IFCG2 run with $FUSE = 20$, which is the configuration that provides the best performance on average, as shown in section 7.1. Figure 4 provides a comparison in terms of speedup considering all 6 CG variants. The x-axis represents the number of cores involved in the parallel run while the y-axis shows the speedups achieved by the different techniques taking the execution time of the Preconditioned CG algorithm on a single core as reference. The experimental setup is described in Section 6.

The most dramatic improvements are achieved when applying IFCG1 and IFCG2 to the *af_shell8* and *cfd2* matrices. For these two matrices IFCG1 running on 16 cores achieves speedups of 10.92x and 10.96x while IFCG2 reaches speedups of 9.72x and 9.62x, respectively. These results are much better than the speedups achieved by the other considered techniques. Indeed, the speedups achieved by the Preconditioned, Pipelined, Gropp and Chronopoulos variants of the CG algorithm are 9.13x, 8.41x, 8.46x and 8.91x in the case of *af_shell8* and 6.41x, 6.63x, 6.66x and 6.8x in the case of *cfd2*, respectively. In the case of the *cfd2* matrix the performance improvements achieved by IFCG1 and IFCG2 are 42.9% and 41.5% better than Chronopoulos, the best state-of-art-technique. IFCG1 provides the highest performance in almost all the cases. The only exception is *ecology2*. In this case, the best speedup on 16 cores is achieved by the Chronopoulos CG (10.98x) although IFCG1 provides a very close speedup of 10.82x when run on 16 cores. This represents a case where techniques proposed in this paper are not better than the state-of-the-art since the input matrix makes the linear system easily scalable (all CG variants achieve speedups close to 10x with respect to PCG running on a single core when solving the *ecology2* on 16 cores).

Besides individual observations, the average speedup over the PCG algorithm running in a single core of both IFCG1 and IFCG2 is significantly better than the one achieved by the other CG versions. Indeed, we can observe from Figure 4 that IFCG1 and IFCG2 reach an average speedup when run on 16 cores of 10.06x and 9.64x, respectively. The other variants achieve speedups of 8.20x (PCG), 8.40x (Pipelined), 8.70x (Gropp) and 8.99x (Chronopoulos) when run on 16 cores. On average, IFCG1 and IFCG2 provide 11.8% and 7.1% performance improvements over the best state-of-the-art technique (Chronopoulos CG). Table 2 lists the iteration counts for all considered matrices and CG variants. Due to the residual check done once every *FUSE* iterations the IFCG1 and IFCG2 algorithms are bound to take more iterations than the other approaches and indeed they take 16 more iterations on average than the other CG variants. This overhead is effectively compensated by overlapping adjacent iterations, as Figure 4 demonstrates.

	PCG	Chronopoulos	Pipelined	Gropp	IFCG	IFCG2
af_shell8	676	676	676	676	680	680
cfd2	563	563	563	563	580	580
ecology2	678	678	678	678	680	680
consph	912	911	926	911	940	940
G2_circuit	430	430	430	430	440	440
G3_circuit	428	428	428	428	480	480
thermal2	2076	2076	2077	2076	2080	2080
Average	794	794	796	794	810	810

Table 2: Iteration counts of all considered methods and matrices. *FUSE* = 20 for IFCG1 and IFCG2

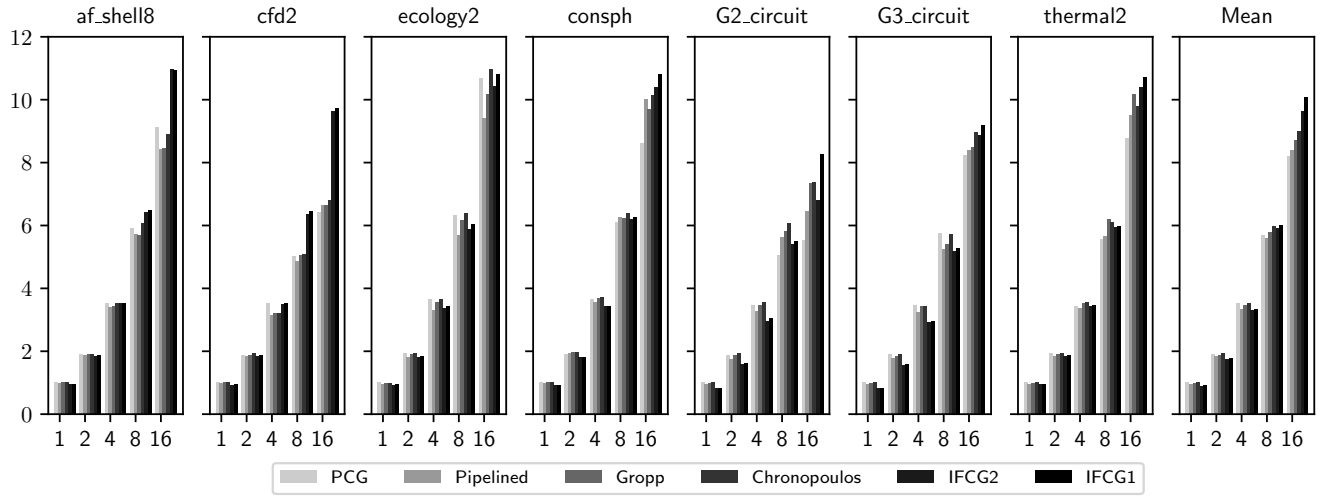

Figure 4: Speedup of all considered CG versions with respect to PCG running on 1 core. The y-axis represents the speedups achieved by the different techniques while x-axis represents core counts.

7.3 Visualizing The Overlap Pattern

The main reason behind the good behavior of IFCG1 and IFCG2 in terms of performance is their capacity to overlap different iterations and this section aims to provide a visual proof of this overlap. Figure 5 displays three 16-core runs composed of 19 iterations. On top of Figure 5 we represent a Pipelined CG execution while IFCG1 is shown in the middle and IFCG2 at the bottom. In the y-axis we represent the 16 threads involved in the parallel run while in the x-axis we show time. The three represented algorithms are applied to the *af_shell8* matrix. In all three views the iterations are marked by distinct colors and all are trimmed for the same time duration (the duration of the pipelined CG since it takes the longest time). The white gaps in Figure 5 represent either idle time or system software activity.

Boundaries between iterations are clearly marked by white gaps in the Pipelined CG representation of Figure 5. Computations belonging to the same iteration are clearly executed in isolation by the Pipelined CG algorithm while this lock-step execution mode is not present in the IFCG1 and IFCG2 representations. For these two cases computations belonging to different iterations are overlapped in a way that only their color identifies the iteration they belong to. There are some small regions represented in white in the IFCG1 and IFCG2 parallel runs that are overlapped with iterations, which account for system software activity. The idle time is almost completely eliminated. The white areas overlapped with the first iteration of IFCG1 and IFCG2 represent computations belonging to previous iterations while the large white areas that appear after the 19 iterations mean that the parallel execution has already finished.

7.4 Tolerance to System Noise

HPC infrastructures frequently get their performance severely degraded by system noise or jitter, which is caused by factors like OS activity, network sharing effects or other phenomena [24]. Although the effects of system jitter may be negligible as long as they are kept at the local scale, parallel operations like reductions or synchronizations are known to strongly amplify its effects by propagating jitter across the whole parallel system [20]. Since algorithms IFCG1 and IFCG2 presented in this paper perform much less reductions or synchronization operations than the Preconditioned CG or the Chronopoulos CG algorithms, they are much more tolerant to jitter effects. To evaluate this additional advantage of IFCG algorithms, this section compares the performance of these 4 algorithms (Preconditioned CG, Chronopoulos CG, IFCG1 and IFCG2) on a noisy regime. The Gropp and Pipelined versions of CG are not considered in this section since, as Figure 4 demonstrates, their behavior is between the one displayed by PCG and Chronopoulos.

We run the 4 algorithms mentioned above on 16 cores considering the input matrices and the experimental setup described in Section 6 and we inject uniformly distributed random noise with an amplitude of 10μs and frequencies of 8kHz and 2kHz. Such noise regimes are close to the measured ones on real systems (1kHz and 25μs [16]) and produce, on average, overheads of 8% ($8 \cdot 10^3 \cdot 10^{-5} = 0.08$) and 2% in sequential computations, respectively. Therefore, any extra overhead suffered by parallel applications under these noise regimes is brought by amplification effects due to parallel synchronization or reduction operations. Parallel executions may also filter out noisy events that take place during idle execution phases.

In Figure 6 we show the elapsed time running on 16 cores of the Preconditioned CG, Chronopoulos CG, IFCG1 and IFCG2 under a noiseless, a 10μs-2kHz and a 10μs-8kHz noise regimes. The y-axis displays the execution time normalized to the Preconditioned CG execution without noise and the x-axis shows the obtained results per matrix plus their average values. On average, the Preconditioned and the Chronopoulos CG algorithms suffer degradation of 19.0% and 14.6% of their execution time, respectively under the 10μs-8kHz noise regime. They are much larger than the 8% degradation expected to be suffered by purely sequential applications, which

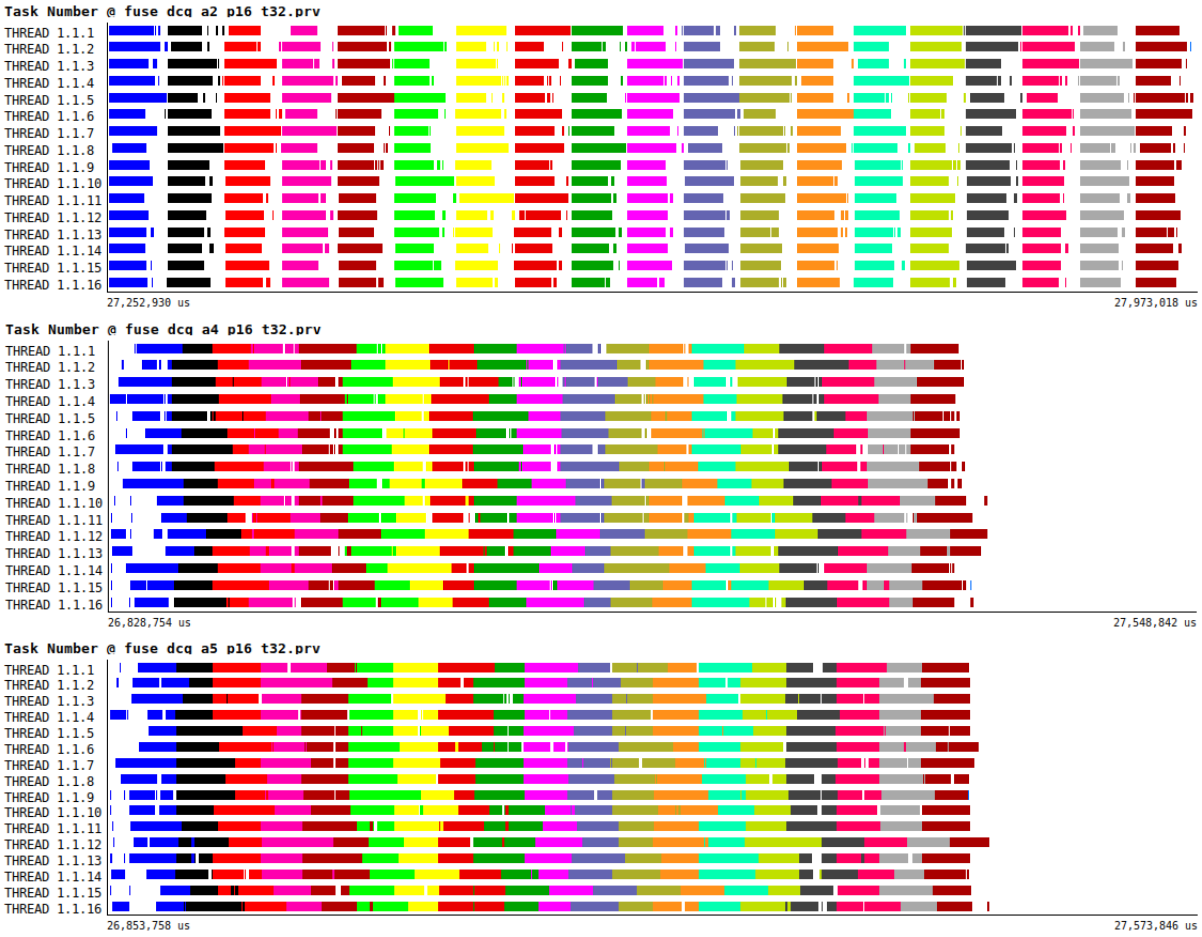

Figure 5: Visualization of 19-iteration runs on 16 cores of Pipelined CG (top), IFCG1 (middle) and IFCG2 (bottom). The input matrix is *af_shell8*

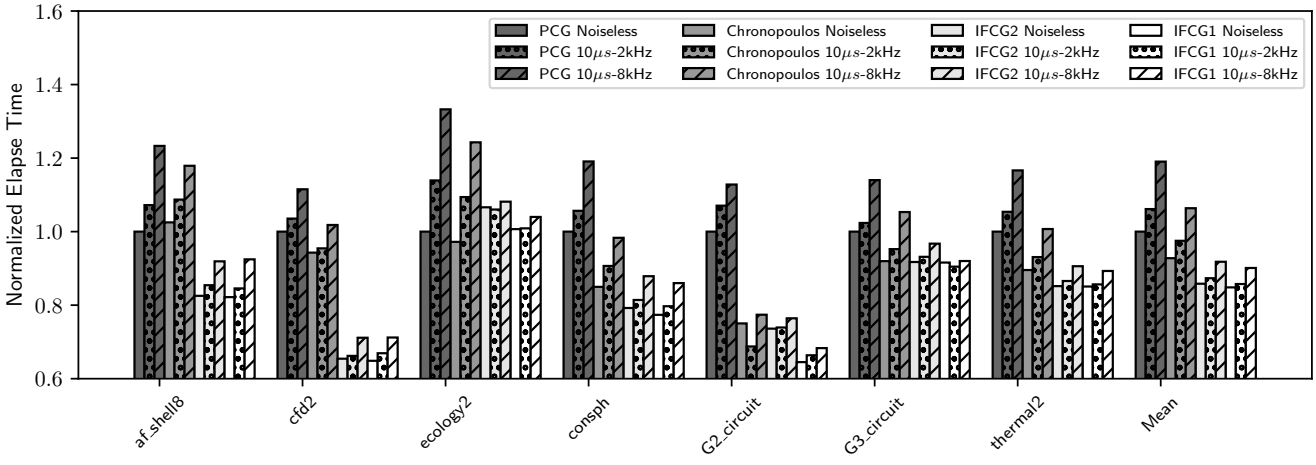

Figure 6: Behavior of different variants of CG running on 16 cores under noiseless, 10μs-2kHz and 10μs-8kHz noise regimes.

implies that noise is amplified by parallel operations like reductions or synchronizations. In contrast, IFCG1 and IFCG2 suffer much milder degradation of just 6.2% and 6.9%, respectively, when exposed to 10μs-8kHz noise. IFCG1 has a 1.18x speedup over Chronopoulos under the 10μs-8kHz, that is, it runs 18.0% faster. Interestingly, both IFCG algorithms run faster under this noisy regime than their state-of-the-art counterparts under the noiseless regimes. In Figure 6 we also show results considering the 10μs-2kHz scenario. For this case, the Preconditioned and the Chronopoulos CG algorithms suffer degradation of 6.1% and 5.1% of their execution time, respectively. In contrast, IFCG1 and IFCG2 suffer milder degradation of just 1.1% and 1.7%, respectively.

8 CONCLUSIONS

This paper presents the IFCG1 and IFCG2 algorithms, which are variants of the CG algorithm where most of the inter-iteration barriers are removed and linear kernels are split into several subkernels. The main difference between IFCG1 and IFCG2 is that the first one aims at hiding parallel reduction costs while the second one avoids idle time by starting the execution of the linear subkernels as soon as possible. The *FUSE* parameter specifies how often both IFCG1 and IFCG2 check for convergence. To maximize the performance of these algorithms the *FUSE* parameter needs to be set up to the optimal value by means of an exhaustive search. This parameter is not input dependent and we find its optimal value to be 20.

To compare the performance of IFCG1 and IFCG2 against other relevant variants of the CG algorithm, we consider 8 matrices from the Florida Sparse Matrix Collection [12] with varying sparsity degrees and dimensions. We find that IFCG1 and IFCG2 achieve significant performance improvements with respect to the state-of-the-art due to their flexibility to overlap computations belonging to different iterations. We also show how reducing the number of global synchronization points makes IFCG1 and IFCG2 much less sensitive to system noise perturbations than their state-of-the-art counterparts. Also, both IFCG1 and IFCG2 display the same numerical stability properties as the most relevant previous techniques.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316) , by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and by the IBM/BSC Deep Learning Center Initiative.

REFERENCES

[1] 2009. *Intel Math Kernel Library Reference Manual.* Intel Corporation.
[2] Z. Bai, D. Hu, and L. Reichel. 1994. A Newton basis GMRES implementation. *IMA J. Numer. Anal.* 14, 4 (1994).
[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM.
[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications *(PACT '08).*
[5] BSC. 2015. Programming Models Group. The Nanos++ Parallel Runtime. https://pm.bsc.es/nanox. (2015).
[6] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Implementing OmpSs support for regions of data in architectures with multiple address spaces *(ICS '13).*
[7] Erin Carson and James Demmel. 2012. *A Residual Replacement Strategy for Improving the Maximum Attainable Accuracy of s-step Krylov Subspace Methods.* Technical Report. University of California, Berkeley.
[8] Erin Carson and James Demmel. 2014. A Residual Replacement Strategy for Improving the Maximum Attainable Accuracy of *s*-Step Krylov Subspace Methods. *SIAM J. Matrix Anal. Appl.* 35, 1 (2014).
[9] A. T. Chronopoulos. 1991. s-Step Iterative Methods for (Non)Symmetric (In)Definite Linear Systems. *SIAM J. Numer. Anal.* 28(6) (1991).
[10] A. T. Chronopoulos and C. W. Gear. 1989. s-Step iterative methods for symmetric linear systems. *J. Comput. Appl. Math.* 25(2) (1989).
[11] Siegfried Cools, Wim Vanroose, Emrullah Fatih Yetkin, Emmanuel Agullo, and Luc Giraud. On Rounding Error Resilience, Maximal Attainable Accuracy and Parallel Performance of the Pipelined Conjugate Gradients Method for Large-scale Linear Systems in PETSc *(EASC '16).*
[12] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38(1) (2011).
[13] Eduardo D'Azevedo, Victor Eijkhout, and Charles Romine. 1993. *LAPACK Working Note 56: Reducing Communication Costs in the Conjugate Gradient Algorithm on Distributed Memory Multiprocessors.* Technical Report. Knoxville, TN, USA.
[14] E. de Sturler and H. A. van der Vorst. 1995. Reducing the Effect of Global Communication in GMRES(M) and CG on Parallel Distributed Memory Computers. *Appl. Numer. Math.* 18, 4 (1995).
[15] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. 1993. Parallel numerical linear algebra. *Acta Numerica* 2 (1993).
[16] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection *(SC '08).*
[17] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. 2013. Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines. *SIAM Journal on Sci. Computing* 35(1) (2013).
[18] W. Gropp. 2010. Update on libraries for blue waters. http://jointlab-pc.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf. (2010).
[19] Exascale Mathematics Working Group(EMWG). 2012. *Applied Mathematics Research for Exascale Computing.* Technical Report. US Department of Energy.
[20] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation *(SC '10).*
[21] M. Hoemmen. 2010. *Communication-avoiding Krylov subspace methods.* Ph.D. Dissertation. University of California.
[22] Wayne Joubert and Graham F. Carey. Parallelizable Restarted Iterative Methods for Nonsymmetric Linear Systems *(SIAM PP '91).*
[23] Gérard Meurant. 1987. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Comput.* 5, 3 (1987).
[24] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. 2011. A Quantitative Analysis of OS Noise *(IPDPS '11).*
[25] OpenMP4.0 2013. Application Program Interface. (2013).
[26] P.Ghysels and W.Vanroose. 2014. Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. *Parallel Comput.* 40 (2014).
[27] John Van Rosendale. 1983. Minimizing inner product data dependencies in conjugate gradient iteration. *ICASE-NASA 172178* (1983).
[28] Yousef Saad. 1984. Practical Use of Some Krylov Subspace Methods for Solving Indefinite and Nonsymmetric Linear Systems. *SIAM J. Sci. Statist. Comput.* 5, 1 (1984).
[29] Henk A. Van Der Vorst and Qiang Ye. 1999. *Residual Replacement Strategies for Krylov Subspace Iterative Methods for the Convergence of True Residuals.* Technical Report.
[30] Laurence Tianruo Yang and Richard P. Brent. The Improved BiCG Method for Large and Sparse Linear Systems on Parallel Distributed Memory Architectures *(PDSECA '02).*
[31] Laurence Tianruo Yang and Richard P. Brent. The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures *(ICA3PP '02).*